SMT-Based Verification of ECMAScript Programs in CPAchecker

Michael Maier

LMU Munich, Germany

JavaScript

- Most commonly used programming language[1]
- Main language for Web applications
- Also used in server, desktop, and mobile applications
- Evaluated by interpreter
- Interpreter define different dialects

[1] https://insights.stackoverflow.com/survey/2019/

ECMAScript

- Specified in standard ECMA-262[2]
- Different standard versions
- Most[3] JS dialects conform to ECMAScript 5.1[4]

---

[2] https://www.ecma-international.org/publications/standards/Ecma-262.htm

[3] http://kangax.github.io/compat-table/es5/

[4] https://www.ecma-international.org/ecma-262/5.1/

Goal

Extend CPAchecker to a restricted subset of ECMAScript 5.1 for SMT based verification approaches

- Parser frontend that creates CFA
- Strongest post operator (SMT formula encoding)

Restricted Subset Of ECMAScript 5.1

- No rarely used statements
- No recursive function calls
- No exceptions in general
- No standard built-in ECMAScript objects

Parser Frontend

Program Representation (CFA)

CFA Operations

- Assumption [`p`]
- Variable declaration `var x` or `var x = e`
- Function declaration `function func(args*) { ... }`
- Assignment `lhs = e`,
- Delete operation `delete o.propName` or `delete o[e]`
- Function call `func(e*)`
- Constructor call `new func(e*)`

SMT Formula Encoding

Strongest Post Operator

- CPAchecker does reachability analysis based on the CFA
- A location is reachable if a path σ to it exists, where $\mathsf{SP}_\sigma(\top)$ is satisfiable
- $\mathsf{SP}_\sigma(\psi) = \mathsf{SP}_{op_m}(\ldots(\mathsf{SP}_{op_i}(\psi))\ldots)$
- Goal of a verification task: Show that an error location is not reachable from the initial program location
- Definition of strongest post operator $\mathsf{SP}_{op}(\psi)$ required

Assumption

Strongest post operator of an assumption [`p`] as operator parameter:

$$\mathsf{SP}_{[p]}(\psi) = \psi \wedge \mathrm{ToBoolean}(p)$$

- $\mathrm{ToBoolean}(p)$ represents large formula
- Other conversions like $\mathrm{ToNumber}$, $\mathrm{ToString}$, etc. have to be defined, too

Dynamic Types

```
var x;
if (predicate) {
  x = 42;
} else {
  x = true;
}
var y = x;
```

- Variables may store values of different types
- SMT variables do not

Value ID

- SMT variables are used as value IDs (integer)[5]
- Associate value and type with value ID using UFs[6]
- Basic idea:
  - `x = 42` is encoded as

$$\text{typeof}(x) = \tau_{\text{number}} \wedge \text{numberValue}(x) = 42$$

  - `x = true` is encoded as

$$\text{typeof}(x) = \tau_{\text{boolean}} \wedge \text{booleanValue}(x) = \top$$

[5] No assignment of specific integer required
[6] UF $=$ uninterpreted function

Static Single-Assingment Form

- Index counter is added for each variable
- Index counter is incremented on assignment
- Fresh value ID is used on every assignment

```
var x;
if (predicate) {
  x = 42;
} else {
  x = true;
}
var y = x;
```

```
var x_0;
if (predicate) {
  x_1 = 42;
} else {
  x_1 = true;
}
var y_0 = x_1;
```

Closure

```
function counter() {
  var x = 0;
  return function next() {
    x = x + 1;
    return x;
  }
}
```

- On each call of `counter`
  - New variable `x` is created
  - New function object `next` is created
- Variable `x` of `counter` is captured by reference from function object `next`

Closure

```
function counter() {
  var x = 0;
  return function next() {
    x = x + 1;
    return x;
  }
}
var c1 = counter();
var c2 = counter();
c1(); // 1
c1(); // 2
c2(); // 1
c1(); // 3
```

- `c1` and `c2` are different instances of `next`
- `c1` and `c2` do **not** capture the same `x`

Scoped Variables

- Use scoped value ID $\mathrm{var}(s, x)$
- s is scope ID (integer)
- x is (unscoped) value ID

Scope Management

- On each function call
  - Create fresh scope ID
  - Put it on *scope stack* of called function object (array of scope IDs of outer function calls)
  - Associate it with this scope stack using UF $\mathrm{scopeStack}$
- Track *current scope* using a variable
- On creation of function object associate it with *current scope*
- Captured variable `x` is encoded as

$$\mathrm{var}(\mathrm{select}(\mathrm{scopeStack}(\mathit{currentScope}), n), x)$$

where n is the nesting level of the declaration of x

String

- String values are mapped to a unique string ID by enumerating all constants of the program

```
var x = "foo";         // "foo" -> 1
var o = { bar: 42 }; // "prop" -> 2
o.foo: 42;             // "foo" -> 1
var t = typeof o;
// result of typeof may be:
//   "undefined" -> 3, "object" -> 4, "boolean" -> 5,
//   "number" -> 6, "string" -> 7, "object" -> 8,
//   "function" -> 9
```

Objects

```
var o = {};
o.foo = 42;
```

- Object value is represented by unique object ID (integer)

$$\mathrm{typeof}(o) \wedge objectValue(o) = 1$$

- Array $objectFields_i$ maps each object ID to its properties

$$\mathrm{store}(objectFields_i, 1, emptyObjectFields)$$

- Property changes tracked by $objectFields$

$$\begin{aligned} fields_{\mathrm{old}} &= \mathrm{select}(objectFields_i, 1) \\ objectFields_{i+1} &= \mathrm{store}(objectFields_i, 1, fields_{\mathrm{new}}) \end{aligned}$$

Properties

- Properties are managed as SMT array
- Property name (string ID) is mapped to value ID
- Names of unset properties are mapped to special value ID $objectFieldNotSet$
- For each set property use fresh value ID p and mark it as set property

$$p \neq objectFieldNotSet$$

Example Of Property Changes

- Map all string IDs of the program to $objectFieldNotSet$[7]

$$\text{store}(\,\text{store}(empty, \text{StringID}(\texttt{"foo"}), objectFieldNotSet),\; \text{StringID}(\texttt{"bar"}), objectFieldNotSet)$$

Program	"foo"	"bar"
`var o = {};`	-	-
`o.foo = 42;`		
`o["foo"] = true;`		
`o.bar = true;`		
`delete o.foo;`		

[7] $objectFieldNotSet$ is displayed as - in table

Example Of Property Changes

- Use fresh value ID p_0 for `o.foo`

$$p_0 \neq \mathit{objectFieldNotSet} \\ \wedge \text{typeof}(p_0) = \tau_{\text{number}} \wedge \text{numberValue}(p_0) = 42$$

Program	"foo"	"bar"
`var o = {};`	-	-
`o.foo = 42;`	p_0	-
`o["foo"] = true;`		
`o.bar = true;`		
`delete o.foo;`		

Example Of Property Changes

- Use fresh value ID p_1 for next property assignment
  `o["foo"] = true`

$$p_1 \neq \mathit{objectFieldNotSet} \wedge \text{typeof}(p_1) = \tau_{\text{boolean}} \wedge \text{booleanValue}(p_1) = \top$$

Program	"foo"	"bar"
`var o = {};`	-	-
`o.foo = 42;`	p_0	-
`o["foo"] = true;`	p_1	-
`o.bar = true;`		
`delete o.foo;`		

Example Of Property Changes

- Use fresh value ID p_2 for next property assignment
  `o.bar = true`

$$p_2 \neq \mathit{objectFieldNotSet} \wedge \mathrm{typeof}(p_2) = \tau_{\mathrm{boolean}} \wedge \mathrm{booleanValue}(p_2) = \top$$

Program	"foo"	"bar"
`var o = {};`	-	-
`o.foo = 42;`	p_0	-
`o["foo"] = true;`	p_1	-
`o.bar = true;`	p_1	p_2
`delete o.foo;`		

Example Of Property Changes

- Use *objectFieldNotSet* as value ID to `delete o.foo;`

Program	"foo"	"bar"
`var o = {};`	-	-
`o.foo = 42;`	p_0	-
`o["foo"] = true;`	p_1	-
`o.bar = true;`	p_1	p_2
`delete o.foo;`	-	p_2

Prototype Chain Example

Object	`"foo"`	`"bar"`	`"foobar"`	`[[Prototype]]` [8]
a	p_0	-	-	b
b	-	p_1	-	c
c	-	p_2	p_3	-

- If property is not found on object recursively look it up on prototype object (if existent)
- Prototype chain $a \to b \to c$

[8] Special string ID for internal prototype property

Prototype Chain

- Prototype chain might be arbitrary long but it is always finite
- We assume that no prototype chain is longer as a maximum $maxPrototypeChainLength$
- Thereby, we can unroll the look-up in the prototype chain
- Drawback: look-up of a property might falsely return `undefined` if $maxPrototypeChainLength$ is too small

Evaluation

Approach

Evaluation of the functional correctness of the implementation of the formula encoding

- Using bounded model checking (BMC) with k-induction
- Based on the test programs of the official ECMAScript Conformance Test Suite *Test262*[9]
- Automatic and manual filtering of test programs that contain unsupported or unimplemented features
- Generate negated tests (negate assertion condition) and check that they fail

[9] `https://github.com/tc39/test262`

Runs

Run	Files	Correct	Incorrect	Unknown
1	780	641	42	97
2	664	662	0	2
3	8625	8593	13	19

Table: Results of different evaluation runs

1. Positive tests after automatic filtering
2. Positive tests after manual filtering and reconfiguring failed tests of 1st run
3. Negative tests of correct tests of 2nd run

Verification Results

- 660 test files correct (21 float encoded as rational)
- 2 test files unknown (timeout)
- 2 bugs[10] found in Test262

[10] https://github.com/tc39/test262/issues/2049

Solved Challenges

- Dynamic types
- Implicit type conversion
- Extensible objects
- Dynamic property access
- Prototype inheritance
- Function objects (higher order functions)
- Closures (scope chain)

Thank you for your attention!

Restricted Subset Of ECMAScript 5.1 (1/2)

- no recursive function calls
- no for-in statements
- no with statement
- no debugger statements
- no exceptions in general
- no standard built-in ECMAScript objects

Restricted Subset Of ECMAScript 5.1 (2/2)

- global variables are not set on the global object
- no arguments object
- it is assumed that all properties are named data properties that are writable and configurable
- implicit function calls from internal methods are not considered
- no regular expression literals
- no `<`, `>`, `<=`, and `>=` to compare strings

CFA Expressions

- binary operators `&`, `/`, `==`, `===`, `>`, `>=`, `in`, `instanceof`, `<<`, `<`, `<=`, `-`, `!==`, `!=`, `|`, `+`, `%`, `>>`, `>>>`, `*`, `^`
- unary operators `+`, `-`, `~`, `!`, `typeof`, `void`
- the property-access operators `o.f` and `o[p]`
- special operator `declaredBy` (not part of regular ECMAScript)

declaredBy Operator

- used to resolve dynamic function calls
- `id declaredBy functionDeclaration` checks if the function object stored in `id` has been declared by the function declaration of `functionDeclaration`

Example of dynamic function call

```
function f() { ... }
function g() { ... }
// ...
r = u();
```

- dynamic function call `u()`
- `f` or `g` might have been assigned to `u`

Resolution of dynamic function call

check by which function declaration the function object `u` has been declared and call that function

```
function f() { ... }
function g() { ... }
// ...
// r = u();
if (u declaredBy f) {
    r = f();
} else if (u declaredBy g) {
    r = g();
} else {
    r = undefined; // this case would throw an exception,
}                  // but exceptions are not covered yet
```

Type Tags

Each type is encoded as a distinct integer called *type tag* (similar to the result of the `typeof` operator):

- $\tau_{\text{undefined}}$
- τ_{boolean}
- τ_{number}
- τ_{string}
- τ_{object}
- τ_{function}

Values

ECMAScript Value	SMT-Formula Encoding Type	Value
Undefined	$\tau_{\text{undefined}}$	single value
Boolean	τ_{boolean}	boolean
Number	τ_{number}	$\text{FP}_{e=11,m=52}{}^{*}$
String	τ_{string}	$\text{FP}_{e=12,m=52}{}^{*}$
Object	τ_{object}	integer
Null	τ_{object}	integer
Function	τ_{function}	integer

* Floating point formula with exponent size e and mantissa size m

String

- string values are mapped to a unique string-ID (floating point number)
- string values that are strict equal[11] have the same ID
- string-IDs are encoded as $\mathrm{FP}_{e=12,m=52}$
- ECMAScript number values are encoded as $\mathrm{FP}_{e=11,m=52}$
- values in range $\mathrm{FP}_{e=11,m=52}$ are used for string representations of their respective ECMAScript number value
- values outside this range are used for all other strings

[11] same sequence of characters

Function Object

- each function object value is encoded like a regular object, but its type is τ_{function}
- its object-ID is associated with its
  - function declaration using an uninterpreted function $\mathrm{declarationOf}$
  - scope using an uninterpreted function $\mathrm{scopeOf}$

Value ID

- Associated type and value UF have to be compatible

$$\text{typeof}(x) = \tau_{\text{boolean}} \wedge \text{numberValue}(x) = 42$$

- Value ID may not be associate with different types

$$\text{typeof}(x) = \tau_{\text{boolean}} \wedge \text{typeof}(x) = \tau_{\text{number}}$$

Value ID

- Different values are associated with different value IDs

$$\begin{aligned} &\mathrm{typeof}(x) = \tau_{\mathrm{number}} \land \mathrm{numberValue}(x) = 42 \\ \land\, &\mathrm{typeof}(y) = \tau_{\mathrm{boolean}} \land \mathrm{booleanValue}(y) = \top \end{aligned}$$

or the same value ID using mutually exclusive conditions

$$\begin{aligned} &(p \land \mathrm{typeof}(x) = \tau_{\mathrm{number}} \land \mathrm{numberValue}(x) = 42) \\ \lor &(\neg p \land \mathrm{typeof}(x) = \tau_{\mathrm{boolean}} \land \mathrm{booleanValue}(x) = \top) \end{aligned}$$

Scope Management

- Create scope ID for $currentScope$ on function call
- Put it on its scope Stack

```
function counter() {
  var x = 0;
  return function next() {
    x = x + 1;
    return x;
  }
}
var c1 = counter();
var c2 = counter();
c1(); // 1
c1(); // 2
c2(); // 1
c1(); // 3
```

$currentScope = 1$

$\text{scopeStack}(1) = (1)$

Scope Management

- Use *currentScope* for local variables of current call

```
function counter() {
  var x = 0;
  return function next() {
    x = x + 1;
    return x;
  }
}
var c1 = counter();
var c2 = counter();
c1(); // 1
c1(); // 2
c2(); // 1
c1(); // 3
```

$$currentScope = 1$$

$$\text{scopeStack}(1) = (1)$$

$$\text{numberValue}(\text{var}(1, x_0)) = 0$$

Scope Management

- Create function object and associate it with $currentScope$

```
function counter() {
  var x = 0;
  return function next() {
    x = x + 1;
    return x;
  }
}
var c1 = counter();
var c2 = counter();
c1(); // 1
c1(); // 2
c2(); // 1
c1(); // 3
```

$$currentScope = 1$$

$$\text{scopeStack}(1) = (1)$$

$$\text{numberValue}(\text{var}(1, x_0)) = 0$$

$$\text{scopeOf}(c1) = 1$$

Scope Management

- Create scope ID for *currentScope* on function call
- Put it on its scope Stack

```
function counter() {
  var x = 0;
  return function next() {
    x = x + 1;
    return x;
  }
}
var c1 = counter();
var c2 = counter();
c1(); // 1
c1(); // 2
c2(); // 1
c1(); // 3
```

$$currentScope = 2$$

$$\text{scopeStack}(1) = (1)$$

$$\text{scopeStack}(2) = (2)$$

$$\text{numberValue}(\text{var}(1, x_0)) = 0$$

$$\text{scopeOf}(c1) = 1$$

Scope Management

- Use *currentScope*, update other scoped variables

```
function counter() {
  var x = 0;
  return function next() {
    x = x + 1;
    return x;
  }
}
var c1 = counter();
var c2 = counter();
c1(); // 1
c1(); // 2
c2(); // 1
c1(); // 3
```

$$currentScope = 2$$
$$\text{scopeStack}(1) = (1)$$
$$\text{scopeStack}(2) = (2)$$
$$\text{numberValue}(\text{var}(1, x_0)) = 0$$
$$\text{numberValue}(\text{var}(2, x_1)) = 0$$
$$\text{var}(1, x_1) = \text{var}(1, x_0)$$
$$\text{scopeOf}(c1) = 1$$

Scope Management

- Simplify formulas

```
function counter() {
  var x = 0;
  return function next() {
    x = x + 1;
    return x;
  }
}
var c1 = counter();
var c2 = counter();
c1(); // 1
c1(); // 2
c2(); // 1
c1(); // 3
```

$$currentScope = 2$$
$$\text{scopeStack}(1) = (1)$$
$$\text{scopeStack}(2) = (2)$$
$$\text{numberValue}(\text{var}(1, x_1)) = 0$$
$$\text{numberValue}(\text{var}(2, x_1)) = 0$$
$$\text{scopeOf}(c1) = 1$$

Scope Management

- Create function object and associate it with *currentScope*

```
function counter() {
  var x = 0;
  return function next() {
    x = x + 1;
    return x;
  }
}
var c1 = counter();
var c2 = counter();
c1(); // 1
c1(); // 2
c2(); // 1
c1(); // 3
```

$$currentScope = 2$$

$$\text{scopeStack}(1) = (1)$$

$$\text{scopeStack}(2) = (2)$$

$$\text{numberValue}(\text{var}(1, x_1)) = 0$$

$$\text{numberValue}(\text{var}(2, x_1)) = 0$$

$$\text{scopeOf}(c1) = 1$$

$$\text{scopeOf}(c2) = 2$$

Scope Management

- Create scope ID for $currentScope$ on function call
- Put it on the scope Stack of the called function object

```
function counter() {
  var x = 0;
  return function next() {
    x = x + 1;
    return x;
  }
}
var c1 = counter();
var c2 = counter();
c1(); // 1
c1(); // 2
c2(); // 1
c1(); // 3
```

$$currentScope = 3$$
$$\text{scopeStack}(1) = (1)$$
$$\text{scopeStack}(2) = (2)$$
$$\text{scopeStack}(3) = (1, 3)$$
$$\text{numberValue}(\text{var}(1, x_1)) = 0$$
$$\text{numberValue}(\text{var}(2, x_1)) = 0$$
$$\text{scopeOf}(c1) = 1$$
$$\text{scopeOf}(c2) = 2$$

Scope Management

- Use $\mathrm{var}(\mathrm{select}(\mathrm{scopeStack}(\mathit{currentScope}), n), x_i)$
- Results in $\mathrm{var}(1, x_i)$

```
function counter() {
  var x = 0;
  return function next() {
    x = x + 1;
    return x;
  }
}
var c1 = counter();
var c2 = counter();
c1(); // 1
c1(); // 2
c2(); // 1
c1(); // 3
```

$\mathit{currentScope} = 3$

$\mathrm{scopeStack}(1) = (1)$

$\mathrm{scopeStack}(2) = (2)$

$\mathrm{scopeStack}(3) = (1, 3)$

$\mathrm{numberValue}(\mathrm{var}(1, x_1)) = 0$

$\mathrm{numberValue}(\mathrm{var}(1, x_2)) = 1$

$\mathrm{numberValue}(\mathrm{var}(2, x_1)) = 0$

$\mathrm{scopeOf}(\mathit{c1}) = 1$

$\mathrm{scopeOf}(\mathit{c2}) = 2$

Variable Declaration

- `var x` is handled like an assignment operation `x = undefined`
- `var x = e` is handled like an assignment operation `x = e`

Function Declaration (1/2)

function declaration `function func(args*) { ... }`

- function object of `func` is created similar to

```
{
    prototype: {},
    length: len
}
```

where `len` represents the count of function parameters

Function Declaration (2/2)

function declaration `function func(args*) { ... }`

- object-ID o of the created object is used in constraints

$$
\begin{aligned}
\mathrm{typeof}(fv) &= \tau_{\mathrm{function}} \\
\mathrm{functionValue}(fv) &= o \\
\mathrm{objectValue}(fv) &= o \\
\mathrm{scopeOf}(o) &= \mathit{currentScope} \\
\mathrm{declarationOf}(o) &= d
\end{aligned}
$$

where

- fv is the scoped variable of the function declaration identifier `func`
- d is the declaration-ID of the declared function

Assignment

different assignment targets:

- assignment to identifier `x = e`
- assignment to object property
  - dot notation `obj.propName = e`
  - bracket notation `obj[propExpr] = e`

Assignment To Identifier

assignment to identifier `x = e`

- associate type of variable with type of expression
- for each type case assign the respective value
- update other scoped variables (same declaration, but different scope-ID)

Assignment To Object Property

`obj.propName = e` or `obj[propExpr] = e`

- create a fresh variable-ID p and mark it as set property (variable)

$$p \neq objectFieldNotSet$$

- assign value to p
- associate property of object with p
- in case of bracket operator, ensure update of `length` property (another property assignment)

Delete Operator

delete operation `delete o.propName` or `delete o[e]`

- equivalent to assigning $objectFieldNotSet$ to the property of the object that is deleted

Function Call

function call `func(e*)`

- execution context switches from the *caller* (function or global code) to the *called* function
- the following has to be done:
  - create a new scope for the called function
  - update current scope stack
  - bind (optional) this argument
  - assign arguments of call to parameter variables of called function

Constructor Call

constructor call `new func(e*)`

- handled like a function call `func(e*)`
- but a new object is created and assigned to the this variable:

```
{
    [[Prototype]]: func.prototype
}
```

where `[[Prototype]]` represents the prototype property

Prototype Chain Example

Object	Property Mapping		
	`'foo'`	`'bar'`	[[Prototype]]
A	-	-	b
A.prototype	foo_2	-	c
B	foo_2	-	-
bProto / B.prototype	foo_2	-	-
b	foo_2	-	-

```
function A() {}
A.prototype.foo = 1;
function B() {}
var bProto = new A();
bProto.bar = 2;
B.prototype = bProto;
var b = new B();
```

Implementation

Options

- **Maximum Field Count:** Required to initially map all properties to $objectFieldNotSet$
- **Maximum Prototype Chain Length:** Required to unroll the prototype chain
- **Usage Of NaN and infinity:**
  - Rational formulas do not support the values NaN and $\pm\infty$ (only as a variable)
  - If floating point formulas are encoded as rational formulas, checking for NaN or $\pm\infty$ can lead to satisfiable and non-tautological formulas
  - Option alters the formula encoding by assuming that those checks always result in $\perp$